[SYSTEM-DESIGN]

Designing Data-Intensive Applications – Chapter 5: Replication

Posted by CHARLES *on* 2020-04-08

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair. —Douglas Adams, Mostly Harmless (1992)

- **Replication** means keeping a copy of the same data on multiple machines that are connected via a network.
  - To keep data geographically close to your users (**reduce latency**)
  - To allow the system to continue working even if some of its parts have failed (**increase availability**)
  - To scale out the number of machines that can serve read queries (**increase read throughput**)
- **All of the difficulty in replication lies in handling changes to replicated data**;
- three popular algorithms for replicating changes between nodes: **single-leader, multi-leader, and leaderless** replication.

Leaders and Followers:

- Each node that stores a copy of the database is called a **replica**;
- How do we ensure that all the data ends up on all the replicas?
- The most common solution for this is called leader-based replication (also known as active/passive or master/slave replication)
  - One of the replicas is designated the **leader** (also known as **master** or **primary**).
  - The other replicas are known as **followers** (read replicas, **slaves**, secondaries, or hot standbys).

    - **Relational DB**: PostgreSQL (since version 9.0), MySQL, Oracle Data Guard [2], and SQL Server's AlwaysOn Availability Groups,
    - **Non-Relational DB**: MongoDB, RethinkDB, and Espresso
    - **Message Broker**: Kafka and RabbitMQ highly available queues
- **Synchronous vs. Asynchronous Replication**: In relational databases, this is often a configurable option; other systems are often hardcoded to be either one or the other.
    - Most database systems apply changes to followers in less than a second.
    - **Synchronous**:
        - Advantages: The follower is guaranteed to have an up-to-date copy of the data that is consistent with the leader.
        - Disadvantages: if the synchronous follower doesn't respond (because it has crashed, or there is a network fault, or for any other reason), the write cannot be processed
        - **impractical** for all followers to be synchronous: any one node outage would cause the whole system to grind to a halt.
    - **Semi-synchronous**: leader and one synchronous follower.
    - **Often**, leader-based replication is configured to be completely **Asynchronous**.
        - a fully asynchronous configuration has the advantage that the leader can continue processing writes, even if all of its followers have fallen behind.
        - especially if there are many followers or if they are geographically distributed.
- **Setting Up New Followers**:
    - How do you ensure that the new follower has an accurate copy of the leader's data?
    - Four Step process: 1, take snapshot; 2, copy snapshot; 3, carry-over new change by using leader's replication log(e.g. log sequence number, and MySQL calls it the binlog coordinates.); 4, after caught up, keep process data from leader;
        - The practical steps of setting up a follower vary significantly by database.
- **Handling Node Outages**: our goal is to keep the system as a whole running despite individual node failures, and to keep the impact of a node outage as small as possible.
    - How do you achieve high availability with leader-based replication?
    - **Follower failure: Catch-up recovery**:
        - from its log, it knows the last transaction that was processed before the fault occurred.
    - **Leader failure: Failover**:
        - **Failover**: one of the followers needs to be promoted to be the new leader, clients need to be reconfigured to send their writes to the new leader, and the other followers need to start consuming data changes from the new leader.
        - **Automatic failover** process usually consists of the following steps:

selection;
    - 3. Reconfiguring the system to use the new leader. The system needs to ensure that the old leader becomes a follower and recognizes the new leader.
  - **Failover is fraught with things that can go wrong**:
    - If asynchronous replication is used, the new leader may not have received all the writes from the old leader before it failed.
    - Discarding writes is especially dangerous if other storage systems outside of the database need to be coordinated with the database contents.
    - In certain fault scenarios, it could happen that two nodes both believe that they are the leader.– aka. **split brain** which will cause conflict data.
    - What is the right timeout before the leader is declared dead?
  - There are no easy solutions to these problems. For this reason, some operations teams prefer to perform failovers manually.
  - These issues—node failures; unreliable networks; and trade-offs around replica consistency, durability, availability, and latency—are in fact fundamental problems in distributed systems.
- **Implementation of Replication Logs**:
  - How does leader-based replication work under the hood?
  - **Statement-based replication(Not Recommended)**:
    (e.g. early version of MySQL)
    - Simplest case, the leader logs every write request (statement) that it executes and sends that statement log to its followers.
    - Potential Issues with this approach:
      - Any statement that calls a nondeterministic function, such as NOW() will get different values;
      - If statements use an auto-incrementing column, or if they depend on the existing data in the database
      - Statements that have side effects (e.g., triggers, stored procedures, user-defined functions)
  - **Write-ahead log (WAL) shipping**: (e.g. PostgreSQL and Oracle)
    - the log is an append-only sequence of bytes containing all writes to the database. Besides writing the log to disk, the leader also sends it across the network to its followers.
    - The main disadvantage is:
      - The log describes the data on a very low level: a WAL contains details of which bytes were changed in which disk blocks. → **closely coupled to the storage engine**.
  - **Logical (row-based) log replication**:
    - An alternative is to use different log formats for replication and for the storage engine, which allows the replication log to be **decoupled from the storage engine internals**.

different storage engines.
    - A logical log format is also easier for external applications to parse.Useful if you want to send the contents of a database to an external system(e.g. **CDC** change data capture)
- **Trigger-based replication**: (e.g. Goldengate for Oracle)
    - if you want to only replicate a subset of the data, or want to replicate from one kind of database to another, or if you need conflict resolution logic then you may need to move replication up to the application layer.
    - An alternative is to use features that are available in many relational databases: triggers and stored procedures.
        - A trigger lets you register custom application code that is automatically executed when a data change (write transaction) occurs in a database system.
    - Trigger-based replication typically has greater overheads than other replication methods, and is more prone to bugs and limitations than the database's built-in replication.
        - But, provide greater flexibility;

Problems with Replication lag:

- Leader-based replication requires all writes to go through a single node, but read-only queries can go to any replica.
- **For Read heavy application**: create many followers, and distribute the read requests across those followers. You can increase the capacity for serving read-only requests simply by adding more followers.
- **Eventual Consistent:** This inconsistency is just a temporary state—if you stop writing to the database and wait a while, the followers will **eventually** catch up and become consistent with the leader.
    - The term "**eventually**" is deliberately vague: in general, there is no limit to how far a replica can fall behind.
- **Reading Your Own Writes**:
    - A user makes a write, followed by a read from a stale replica. To prevent this anomaly, we need **read-after-write consistency**.
    - It makes no promises about other users: other users' updates may not be visible until some later time. However, it reassures the user that their own input has been saved correctly
    - How can we implement read-after-write consistency in a system with leader-based replication? e.g.
        - When reading something that the user may have modified, read it from the leader; otherwise, read it from a follower.
        - You could track the time of the last update and, for one minute after the last update, make all reads from the leader;
        - The client can remember the timestamp of its most recent write;

on one device and then views it on another device, they should see the information they just entered.
    - If your approach requires reading from the leader, you may first need to route requests from all of a user's devices to the same datacenter.
- **Monotonic Reads**:
    - when reading from asynchronous followers is that it's possible for a user to see things moving backward in time. This can happen if a user makes several reads from different replicas.
    - **Monotonic reads**: It's a lesser guarantee than strong consistency, but a stronger guarantee than eventual consistency.
        - One way of achieving monotonic reads is to make sure that each user always **makes their reads from the same replica**.
- **Consistent Prefix Reads**:
    - violation of causality. E.g. If some partitions are replicated slower than others, an observer may see the answer before they see the question.
    - **consistent prefix reads**: This guarantee says that if a sequence of writes happens in a certain order, then anyone reading those writes will see them appear in the same order.
        - This is a particular problem in partitioned (sharded) databases,
- **Solutions for Replication Lag**:
    - Stronger guarantees;
    - **Transactions**: they are a way for a database to provide stronger guarantees so that the application can be simpler.
    - **Single-node transactions** have existed for a long time, but not for distributed systems.

Multi-Leader Replication:

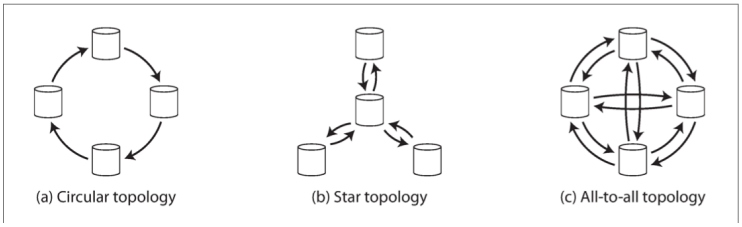
- Leader-based replication has one major downside: there is only one leader, and all writes must go through it. → Single point of failure;
- **Multi-leader configuration** (also known as master–master or active/active replication): A natural extension of the leader-based replication model is to allow more than one node to accept writes.
    - Replication the same: each node that processes a write must forward that data change to all the other nodes.
- **Use Cases for Multi-Leader Replication**:
    - It rarely makes sense to use a multi-leader setup within a single datacenter, because the benefits rarely outweigh the added complexity.
    - Some databases support multi-leader configurations by default, but it is also often implemented with external tools. (e.g. Tungsten Replicator for MySQL, BDR for PostgreSQL, and GoldenGate for Oracle)
    - **Multi-datacenter operation**:
        - In a multi-leader configuration, you can have **a leader in each datacenter**. Each datacenter's leader replicates its changes to the leaders in other

    - Tolerance of datacenter outages:
    - Tolerance of network problems;
      - A multi-leader configuration with asynchronous replication can usually tolerate network problems better: a temporary network interruption does not prevent writes being processed.
  - **Downside**:
    - the same data may be concurrently modified in two different datacenters, and those **write conflicts** must be resolved.
    - As multi-leader replication is a somewhat retrofitted feature in many databases, there are often subtle configuration pitfalls and surprising interactions with other database features.
  - **Clients with offline operation**: E.g. calendar, mobile devices etc. (CouchDB)
    - In this case, every device has a local database that acts as a leader (it accepts write requests), and there is an asynchronous multi-leader replication process (sync) between the replicas of your calendar on all of your devices.
    - multi-leader replication is a tricky thing to get right.
  - **Collaborative editing**:
    - Real-time collaborative editing applications allow several people to edit a document simultaneously. E.g. Etherpad, Google Docs.
    - For faster collaboration, you may want to make the unit of change very small (e.g., a single keystroke) and avoid locking.
- **Handling Write Conflicts**:
  - The **biggest problem** with multi-leader replication is that write conflicts can occur, which means that conflict resolution is required.
  - **Synchronous versus asynchronous conflict detection**:
    - the conflict is only detected asynchronously at some later point in time
    - If you want synchronous conflict detection, you might as well just use single-leader replication.
  - **Conflict avoidance**:
    - The simplest strategy for dealing with conflicts is to **avoid** them: if the application can ensure that all writes for a particular record go through the same leader, then conflicts cannot occur.
  - **Converging toward a consistent state**:
    - the database must resolve the conflict in a **convergent** way, which means that all replicas must arrive at the same final value when all changes have been replicated.
    - **Ways of achieving convergent conflict resolution**:
      - Give **each write a unique ID** (e.g., a timestamp, a long random number, a UUID, or a hash of the key and value), pick the write with the highest ID as the winner, and throw away the other writes. (**LWW — Last Write Wins**)
        - This approach is popular, it is dangerously prone to data loss;
      - Give **each replica a unique ID** (also potential data loss);

some later time (e.g. prompt user)

- **Custom conflict resolution logic**:
  - Most multi-leader replication tools let you write conflict resolution logic using application code.
    - On Write: As soon as the database system detects a conflict in the log of replicated changes, it calls the conflict handler. (e.g. Bucardo)
    - On Read: When a conflict is detected, all the conflicting writes are stored. The next time the data is read, these multiple versions of the data are returned to the applica- tion. (e.g. CouchDB)
- **Automatic Conflict Resolution**:
  - **Conflict-free replicated data** types (CRDTs, two-way merge) (e.g. Riak 2.0)
  - **Mergeable persistent data** structures (similar to git, perform three-way merge)
  - Operational transformation: It was designed particularly for concurrent editing of an ordered list of items, such as the list of characters that constitute a text document. (Etherpad, Google Docs)

- **Multi-Leader Replication Topologies**:
  - A replication topology describes the communication paths along which writes are propagated from one node to another.



  - The most general topology is all-to-all, in which every leader sends its writes to every other leader.
  - In circular and star topologies, a write may need to pass through several nodes before it reaches all replicas.
    - Problem: if just one node fails, it can interrupt the flow of replication messages between other nodes, causing them to be unable to communicate until the node is fixed.
  - Fault tolerance of a more densely connected topology (such as all-to-all) is better because it allows messages to travel along different paths, avoiding a single point of failure.
    - Problem: some network links may be faster than others (e.g., due to network congestion), with the result that some replication messages may “overtake” others.

Leaderless Replication:

- Amazon used it for its in-house **Dynamo system**.vi Riak, Cassandra, and Voldemort are open source datastores with leaderless replication models inspired by

replicas, while in others, a **coordinator** node does this on behalf of the client.

- **Writing to the Database When a Node Is Down**:
  - In a leaderless configuration, failover does not exist.
  - A **quorum write, quorum read, and read repair** after a node outage.
  - When a client reads from the database, it doesn't just send its request to one replica: **read requests are also sent to several nodes in parallel**.
  - **Read repair and anti-entropy**:
    - The replication scheme should ensure that eventually all the data is copied to every replica.
    - **Read repair**: When a client makes a read from several nodes in parallel, it can detect any stale responses. (when read is heavy)
    - **Anti-entropy process**: In addition, some datastores have a background process that constantly looks for differences in the data between replicas and copies any missing data from one replica to another. (maybe delay)
  - **Quorums for reading and writing**:
    - As long as **w + r > n**, we expect to get an up-to-date value when reading, because at least one of the r nodes we're reading from must be up to date.
    - **Reads and writes that obey these r and w values are called quorum reads and writes**:
      - You can think of **r** and **w** as the minimum number of votes required for the read or write to be valid.
    - In Dynamo-style databases, the parameters n, w, and r are typically configurable. A common choice is to make **n** an odd number (typically 3 or 5) and to set w = r = (n + 1) / 2 (rounded up).
    - The quorum condition, w + r > n, allows the system to tolerate unavailable nodes as follows:
      - If w < n, we can still process writes if a node is unavailable.
      - If r < n, we can still process reads if a node is unavailable.
      - With n = 3, w = 2, r = 2 we can tolerate one unavailable node.
      - With n = 5, w = 3, r = 3 we can tolerate two unavailable nodes.
      - The parameters w and r determine how many nodes we wait for
    - If fewer than the required w or r nodes are available, writes or reads return an error.
- **Limitations of Quorum Consistency**:
  - If you have **n replicas**, and you choose w and r such that w + r > n, you can generally expect every read to return the most recent value written for a key.
    - because the set of nodes to which you've written and the set of nodes from which you've read must overlap
  - Often, r and w are chosen to be a majority (more than n/2) of nodes, because that ensures w + r > n while still tolerating up to n/2 node failures.
  - With a smaller w and r:
    - more likely to read stale values

- Even with w + r > n, there are many edge cases that could cause problems
  - Thus, although quorums appear to guarantee that a read returns the latest written value, in practice it is not so simple.
- Dynamo-style databases are generally optimized for use cases that can tolerate **eventual consistency**.
  - Stronger guarantees generally require transactions or consensus.
- **Monitoring staleness**:
  - For leader-based replication, the database typically exposes metrics for the replication lag, which you can feed into a monitoring system.
  - In systems with leaderless replication, there is no fixed order in which writes are applied, which makes monitoring more difficult.
  - Eventual consistency is a deliberately vague guarantee, but for operability it's important to be able to quantify "eventual."

- **Sloppy Quorums and Hinted Handoff**:
  - Databases with appropriately configured quorums can tolerate the failure of individual nodes without the need for failover.
  - Leaderless replication appealing for use cases that require high availability and low latency, and that can tolerate occasional stale reads.
  - **Sloppy quorum**: writes and reads still require w and r successful responses, but those may include nodes that are not among the designated n "home" nodes for a value. (e.g. Riak enabled default, Cassandra, Voldemort disabled)
    - particularly useful for increasing write availability: as long as any w nodes are available, the database can accept writes.
    - isn't a quorum at all in the traditional sense. It's only an assurance of durability.
  - **Hinted handoff**: Once the network interruption is fixed, any writes that one node temporarily accepted on behalf of another node are sent to the appropriate "home" nodes.
  - **Multi-datacenter operation**:
    - **Cassandra** and Voldemort implement their multi-datacenter support within the normal leaderless model: the number of replicas n includes nodes in all datacenters, and in the configuration you can specify how many of the n replicas you want to have in each datacenter.
    - **Riak** keeps all communication between clients and database nodes local to one datacenter, so n describes the number of replicas within one datacenter.
    - Cross-datacenter replication between database clusters happens asynchronously in the background, in a style that is similar to multi-leader replication
  - **Detecting Concurrent Writes**:
    - **Dynamo-style databases** allow several clients to concurrently write to the same key, which means that conflicts will occur even if strict quorums are used.

    - declare that each replica need only store the most "recent" value and allow "older" values to be overwritten and discarded.
    - writes are concurrent, so their order is undefined.
    - Even though the writes don't have a natural ordering, we can force an arbitrary order on them.
    - LWW achieves the goal of eventual convergence, but at the cost of durability
        - If losing data is not acceptable, LWW is a poor choice for conflict resolution.
- **The "happens-before" relationship and concurrency**:
    - An operation A happens before another operation B if B knows about A, or depends on A, or builds upon A in some way.
    - **Concurrency, Time, and Relativity**:
        - For defining **concurrency**, exact time doesn't matter: **we simply call two operations concurrent if they are both unaware of each other, regardless of the physical time at which they occurred.**
- **Capturing the happens-before relationship**:
    - server can determine whether two operations are concurrent by looking at the **version numbers**—it does not need to interpret the value itself (so the value could be any data structure).
    - When a write includes the version number from a prior read, that tells us which previous state the write is based on.
- **Merging concurrently written values**:
    - if several operations happen concurrently, clients have to clean up afterward by merging the concurrently written values. (aka siblings in Riak)
        - Merging sibling values is essentially the same problem as conflict resolution in multi-leader replication
    - a reasonable approach to merging siblings is to just take the union.
- **Version vectors**: (C: allow data to diverge and converge,like git)
    - We need to use a version number per replica as well as per key.
    - Each replica increments its own version number when processing a write, and also keeps track of the version numbers it has seen from each of the other replicas.
    - **Version vector**: The collection of version numbers from all the replicas.
        - most interesting is probably the dotted version vector, which is used in Riak 2.0
    - The version vector allows the database to distinguish between overwrites and concurrent writes.
    - The version vector structure ensures that it is safe to read from one replica and subsequently write back to another replica.
    - **Version vectors and vector clocks**:

data structure to use.

Summary:

- **Replication** can serve several purposes:
  - **High availability**: Keeping the system running, even when one machine (or several machines, or an entire datacenter) goes down
  - **Disconnected operation**: Allowing an application to continue working when there is a network interruption
  - **Latency**: Placing data geographically close to users, so that users can interact with it faster
  - **Scalability**: Being able to handle a higher volume of reads than a single machine could handle, by performing reads on replicas
- Despite being a simple goal, keeping a copy of the same data on several machines' replication turns out to be a remarkably tricky problem.
- Three main approaches to replication:
  - **Single-leader** replication
    - Clients send all writes to a single node (the leader), which sends a stream of data change events to the other replicas (followers). Reads can be performed on any replica, but reads from followers might be stale.
  - **Multi-leader** replication
    - Clients send each write to one of several leader nodes, any of which can accept writes. The leaders send streams of data change events to each other and to any follower nodes.
  - **Leaderless** replication
    - Clients send each write to several nodes, and read from several nodes in parallel in order to detect and correct nodes with stale data.
- Each approach has advantages and disadvantages.
  - Single-leader:
    - s popular because it is fairly easy to understand and there is no conflict resolution to worry about.
  - Multi-Leader & Leaderless:
    - Can be more robust in the presence of faulty nodes, network interruptions, and latency spikes—at the cost of being harder to reason about and providing only very weak consistency guarantees.
- Replication can be **synchronous** or **asynchronous**, which has a profound effect on the system behavior when there is a fault.
  - It's important to figure out what happens when replication lag increases and servers fail.
- Consistency models which are helpful for deciding how an application should behave under replication lag:
  - Read-after-write consistency;
  - Monotonic reads;
  - Consistent prefix reads;